Automatic Music Teaching: A Logic-Probabilistic Étude Generator

Vincent Nys Jon Sneyers Danny De Schreye

Department of Computer Science, KU Leuven

Abstract

We explore the potential of automatic music generation as a tool in music teaching, to create musical études aimed at improving sight-reading or technique. The logic-probabilistic music generation system APOPCALEAPS can be parameterized to generate études in a particular style, but does not take into account the crucial didactic notion of difficulty. We describe a modified version of APOPCALEAPS which generates progressively more difficult pieces. As a case study, we restrict ourselves to sight-reading études in the blues style, for electric guitarists. Experiments show that our approach is very promising and that it correctly captures the notion of difficulty.

1 Introduction

Automatic generation of music is a topic which has been studied in many branches of artificial intelligence. Different paradigms, such as Markov models [1], generative grammars [6] and genetic algorithms [2] have been used to generate music. The current work presents an approach based on probabilistic programming with constraints. Its objective was to generate exercises tailored to an individual musician. In this way, students struggling with their curriculum can build up to the required material, while students who are ahead of their peers can be challenged. A similar goal is set in [8] and in [7]. These systems offer many degrees of freedom, but they focus on relatively low-level aspects of music (see Section 6).

To demonstrate the feasibility of our approach, we have developed a system which generates sheet music for blues guitarists. This system builds on APOPCALEAPS (Automatic Pop Composer And Learner of Parameters), a music generator detailed in [12]. Extensions which were added to APOPCALEAPS consist of: a way to parameterize the perceived difficulty of rhythm and melody, a system of abstract diatonic note functions which allows for musical themes with variation and a set of modifications to obtain output which is clearly recognizable as blues music. This proof-of-concept was successfully evaluated by sight-reading musicians. Some representative examples of the system's output are given in Figure 1 and more samples, with MIDI output, can be found at `https://perswww.kuleuven.be/~u0055408/samples-apopcaleaps.html`.

Section 2 discusses metrics used to evaluate the difficulty of a piece of music. An introduction to CHRiSM and APOPCALEAPS is provided in Section 3. The different extensions made to the APOPCALEAPS system are presented in Section 4. Section 5 contains an experimental evaluation of the prototype. We wrap up with avenues for future work and conclusions.

2 Musical Difficulty

We briefly introduce some musical concepts, before discussing the metrics used to evaluate musical difficulty. For more details about music, see [5].

Note lengths. A note has an audible frequency or "pitch" and a "length". The length of a note describes its duration. It is expressed as a fraction, and is compared to the length of a beat. For instance, if a "fourth" meter is used, a "quarter" note has a duration of one beat. In practice, most notes are

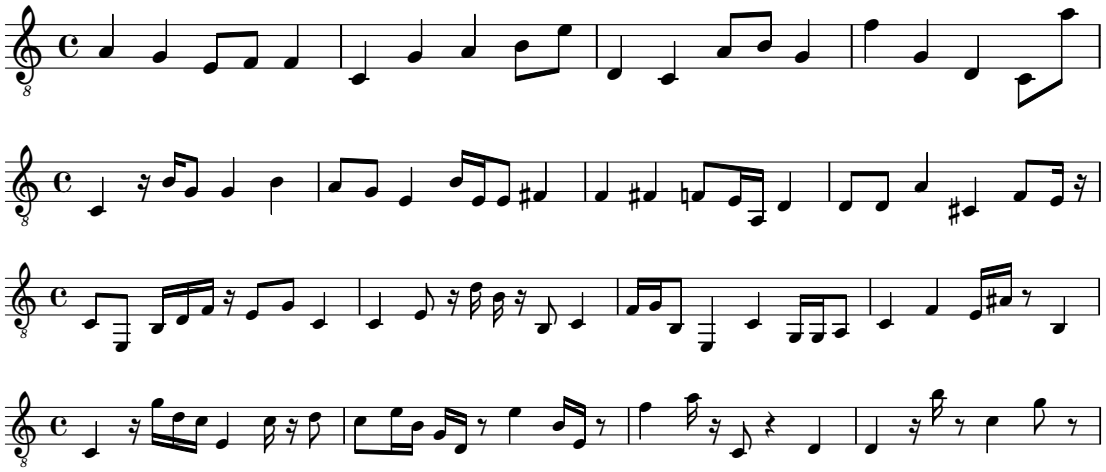

Figure 1: Four four-measure excerpts, from lowest (top) to highest (bottom) difficulty.

binary fractions and anything shorter than a $\frac{1}{32}$ note is rare. Whenever note lengths are mentioned in this discussion, it is assumed that the "common" meter ($\frac{4}{4}$) is used.

Intervals. An interval represents the distance between two notes, i.e. the ratio of their frequencies. Specific intervals invoke a certain mood: a "minor third" interval sounds sad or ominous, while a "major third" interval has an uplifting sound.

Scales. Modulo octaves, there are 12 distinct pitches in Western music. A scale is a sequence of typically 5 to 8 pitches. Scales provide a framework to build melodies. If notes are chosen at random, the result is chaotic. If a commonly known scale is used, the result will be perceived as more coherent.

Difficulty. The difficulty of a piece of music is a subjective notion. Literature on the subject of sight-reading [10] states that "the easiest way to enhance the information processing capacity would be to practice pattern recognition and chunking of note events". The difficulty of a piece is therefore equated to its likelihood. However, even defining the likelihood of a musical pattern is a non-trivial problem.

To take into account the strengths and weaknesses of a music student, we consider the likelihood of musical patterns along two axes: melody and rhythm. We therefore refer to the statistical likelihood of a melody as its "melodic difficulty" and to that of a rhythm as its "rhythmic difficulty". Our goal is to generate compositions whose melodic and rhythmic difficulty each fall within a pre-specified range. In what follows, we will refer to an algorithm which determines difficulty as a *metric*.

To find the likelihood of particular patterns, a corpus of blues samples was obtained. Because there is no structured digital corpus of blues music, we wrote a web crawler which looks for songs performed by well-known musicians on [15]. This yielded 596 songs, from which patterns were learned. Only monophonic lead voice sections were used; polyphony was ignored for simplicity.

2.1 Rhythmic Difficulty

To obtain a metric for rhythmic difficulty, a Markov model was generated from the corpus. Standard Markov models can be summarized as finite state machines with probabilistic transitions. They are discussed in more detail in [9]. The model represents a state using three elements of a rhythmic context: the current beat, the duration of the previous element, a boolean value indicating whether the previous element was a rest. Figure 2 shows a sequence generated from the model. Rhythms were isolated by dividing each song into measures and distinguishing between sounds and rests. The metric itself is the likelihood of a given rhythm according to the Markov model.

The entire corpus was used to parameterize the model. The metric was not tested against a separate set, because there is no corpus with objective annotations for difficulty. It was, however, evaluated by

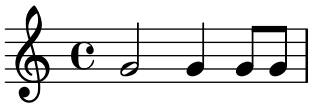

Figure 2: Illustration of the Markov model for rhythm. This sequence is represented as [(1, 2, True), (3, 1, True), (4, $\frac{1}{2}$, True), ($4\frac{1}{2}$, $\frac{1}{2}$, True)].

musicians. As Section 5 shows, this simple metric captures the notion of rhythmic difficulty very well.

2.2 Melodic Difficulty

We restrict melodic difficulty to the likelihood of a sequence of pitches. Intuitively, sequences with high likelihood are short or contain subsequences which should be familiar to musicians and can be read as "chunks", as they are referred to in [10]. For this likelihood, a satisfactory metric was found in an off-the-shelf formula for the predictability of melodies discussed in [14]. This metric is based on a number of musical rules of thumb, formulated as probability distributions. It takes into account the following aspects: the a priori probability of a musical key and average pitch, the limited deviation from the average pitch, the typical frequency of particular melodic intervals and the proximity between sequential notes.

Temperley's data are drawn from the Essen Folksong Collection[3], a corpus of folk songs. Ideally, the data would consist of blues music. However, our blues corpus lacks required annotations, which currently cannot be added algorithmically. Fortunately, the principles behind Temperley's metric apply to most of Western music. The experiments in Section 5 confirm this.

3 CHRiSM and APOPCALEAPS

CHRiSM (CHance Rules induce Statistical Models) is a programming language which combines CHR [4] with PRISM [11]. An introduction to CHRiSM can be found in [13]. The current section summarizes this introduction. A CHRiSM program $\mathcal{P}$ consists of a sequence of *chance rules*. Chance rules rewrite a multiset $\mathbb{S}$ of data elements, which are called *constraints*. Syntactically, a constraint *c(*X_1*,..,*X_n*)* looks like a Prolog predicate: it has a functor *c* of arity n and arguments $X_1,..,X_n$ which are Prolog terms. The multiset $\mathbb{S}$ of constraints is called the *store*. The initial store is called the *query* or *goal*, the final store (after exhaustive rule application) is called the *answer* or *result*.

A chance rule is of the following form: "*P ?? Hk* $\setminus$ *Hr* <=> *G* | *B*." where *P* is a probability expression (an explicit number or an *experiment name*), *Hk* is a conjunction of (kept head) constraints, *Hr* is a conjunction of (removed head) constraints, *G* is an optional guard condition (a PRISM goal to be satisfied), and *B* is the body of the rule. The body *B* can contain both constraints and PRISM goals.

Intuitively, the meaning of a chance rule is as follows: If the store $\mathbb{S}$ contains elements that match the head of the rule and the guard *G* is satisfied, we can consider rule application. The subset of $\mathbb{S}$ that matches the head of the rule is called a rule *instance*. With a probability referred to by *P*, the rule instance leads to a rule application. Every rule instance may only be considered once. Rule application has the following effects: the constraints matching *Hr* are removed from the store (those matching *Hk* are kept), and the body *B* is executed, i.e. PRISM goals are called and constraints are added to the store.

Since CHRiSM is based on CHR(PRISM), all features of PRISM can be used in the body of chance rules. In particular, the PRISM built-in *msw(E,V)* can be used to randomly sample experiment *E* and unify *V* with its outcome value. The outcome space of experiment *E* has to be declared using *values(E,VL)*, where *VL* is a (finite) list of ground Prolog terms. Probability distributions can be defined manually using *set_sw(E,PL)*, where *PL* is a list of probabilities (one for every outcome value), or they can be automatically learned from a training set.

APOPCALEAPS. The core of APOPCALEAPS [12] is a CHRiSM program. As input, it takes a high-level specification of the piece to be generated: instruments and their ranges, meter and number of measures to generate, etc. It produces an internal representation as CHRiSM-constraints, which is converted to a PDF music sheet and MIDI file.

To illustrate, an example input specification could be as follows:

```
meter(4,4), key(minor), tempo(120), measures(13), voice(melody),
instrument(melody,'acoustic guitar (nylon)'), set_range(melody,c,3,-8,28),
shortest_duration(melody,16), max_jump(melody,12), max_repeat(melody,3)
```

Music generation in APOPCALEAPS starts by determining the overall chord sequence. Then, the rhythm is generated, starting from a simple metronome-like rhythm, whose beats are subsequently split, recursively and probabilistically. Pitches are then assigned, taking into account the beat position and underlying chord. In Section 4.3 we will discuss pitch assignment in more detail. To allow for syncopation, consecutive identical notes are probabilistically tied together. Finally, the output files are produced.

4 Étude Generation

Our prototype modifies and extends the existing APOPCALEAPS system in various ways. Since we did not have the resources to manually create a suitable training set, we have simply used the generic default probability distributions. For most of the input specification, such as the time signature and tempo, we used hard-coded genre-specific parameters adapted to blues music.

We also replaced the standard chord generation of APOPCALEAPS with a set of fixed chord progressions. Chord progressions are essential to the structure of a composition. In blues music, especially, chord progressions play an important part: the "12-bar progression" underlies many blues classics. The fixed progressions we provided are all 12-bar progressions typical of the blues genre.

4.1 Achieving the Desired Difficulty

A naive way to get an étude of a given difficulty would be to simply generate a candidate piece of music, evaluate its difficulty, and try again if the difficulty is not right. Given that we are aiming for études containing 12 measures, such an approach would be extremely inefficient.

Ideally, generating and testing would be perfectly intertwined. However, that proved rather hard to do. It would require the difficulty evaluators, which were implemented in different languages, to work directly on intermediate representations. Instead, we modified the original APOPCALEAPS system to achieve a limited form of intertwining. Essentially, we exploit the fact that CHR (and thus CHRiSM) blurs the boundaries between code and data, lending itself well to so-called *anytime* algorithms. Such algorithms still yield (partial or suboptimal) solutions if they are prematurely interrupted. We evaluate and reuse the intermediate representation of a partially generated piece of music as part of a new input specification, in an iterative generation process where generation and testing are interleaved.

We had to adapt APOPCALEAPS so it could work with partially generated pieces. The additional input constraint `unspecified_measure/1` indicates which measures actually still have to be generated; other measures are given as part of the input query in terms of the intermediate representation (`beat/5`, `note/5`, etc.). The composition can in this way be created and verified one measure at a time by gradually removing `unspecified_measure/1` constraints from the input specification.

Some technical modifications had to be made in order to deal with this iterative approach. In general, steps which instantiate concrete musical structures in measures are skipped. The phases in which they are performed begin with the first unspecified measure instead of the first measure, they end with the last unspecified measure instead of the last measure and they move from one measure to its "unspecified successor" rather than to its numeric successor. The following rules illustrate this approach:

```
unspecified_successor(0,X) \ make_measures(0) <=> phase(split_beats(X)).
max_unspecified(M) \ phase(split_beats(M)) <=> phase(make_notes).
unspecified_successor(M,N) \ phase(split_beats(M)) <=> phase(split_beats(N)).
```

In addition to skipping generation stages, the modified program avoids integrity checks for prespecified measures. For instance, the range of an instrument is not checked for a measure which has already been generated.

Another procedure in APOPCALEAPS which requires modification concerns note joining. Notes with the same pitch are probabilistically merged in the original program. Inside a single measure, this is easily avoided by only considering unspecified measures. In the case of adjacent measures, however, this is only allowed when *both* measures are unspecified, as joining would alter a fully specified measure.

4.2 Difficulty Calibration

The probability of a pitch sequence or rhythm has no significance to an end user. Instead, musicians think in terms of "levels": music schools grade the overall difficulty of pieces of music and students follow a path from the first to the last level. The system described here applies the same principle by generating a large amount of sample output and by clustering pieces of similar difficulty.

To perform this calibration, 2006 compositions were generated without restrictions on their likelihood. 1000 of these were generated in the major mode and 1006 were generated in the minor mode. Each composition consisted of 12 measures. For each of these 24072 individual measures, the rhythmic and melodic likelihood were determined. Using these absolute figures, quartile boundaries for each metric were established. With these boundaries, the rhythm and melody of a given measure can be assigned to a quartile, which approximates the idea of a level of difficulty.

4.3 Abstract Notes

We have made an additional modification to APOPCALEAPS in order to better capture the typically repeating structure of études — and of music in general. We first discuss a modification to the way note pitches are assigned and then explain how this enables repetition with variation.

The assignment of note pitches now occurs in two stages. In the first stage, an "abstract note" is assigned. Abstract notes do not represent absolute pitches, but rather diatonic note functions with respect to the current chord in the progression. The following abstract notes are defined (example concretizations w.r.t. A minor):

- "**t**onic" (A), "**m**ediant" (C), "**d**ominant" (E): together, these three abstract notes correspond to the chord notes of the current underlying chord
- "**s**cale" (B, D, F, G): any other scale note (w.r.t. the diatonic key of the piece)
- "**n**onscale" (C♯ / D♭, D♯ / E♭, F♯ / G♭, G♯ / A♭, A♯ / B♭): any other note
- "**a**pproach": this is a "scale" note with additional constraints: it is followed by a nearby note which is either another approach note, or a chord note
- "**b**lues" (D, D♯ / E♭, G): notes on the blues scale (besides the chord notes)

In the second stage of pitch assignment, these abstract notes are then instantiated to concrete notes. This instantiation can either be deterministic (if there is only one possible note value, like for 'tonic') or it can be another probabilistic choice.

Originally, APOPCALEAPS only used scale notes; chord notes and other scale notes were only implicitly distinguished through the probability distributions. Non-scale notes, and in particular "blues" notes, are crucial to get a better match with the blues genre. The following CHRiSM code fragment (slightly simplified for brevity) shows part of the new rules for note pitch assignment.

```
% approach notes should be followed by other approach notes or chord notes
anote(V,M1,N1,X1,approach), next_beat(V,M1,N1,X1,M,N,X), anote(V,M,N,X,Next)
   ==> member(Next,[approach,tonic,mediant,dominant]).

[...]

make_notes_measure(M), beat(V,M,N,X,D), mchord(M,C)
   ==> abstract_beat(M,N,X,AB) |
       soft_msw(note_choice(V,AB),ANote),   msw(concrete(V,C,ANote),Note),
       anote(V,M,N,X,ANote),                note(V,M,N,X,Note).
```

The first rule enforces constraints on "approach" notes. The second rule probabilistically assigns first abstract notes and then concrete note pitches. If these assignments end up violating one of the constraints, then CHRiSM will backtrack over both the non-deterministic choices (e.g. the `member/2` in the first rule) and the soft-probabilistic choices (e.g. the `soft_msw/2` in the second rule).

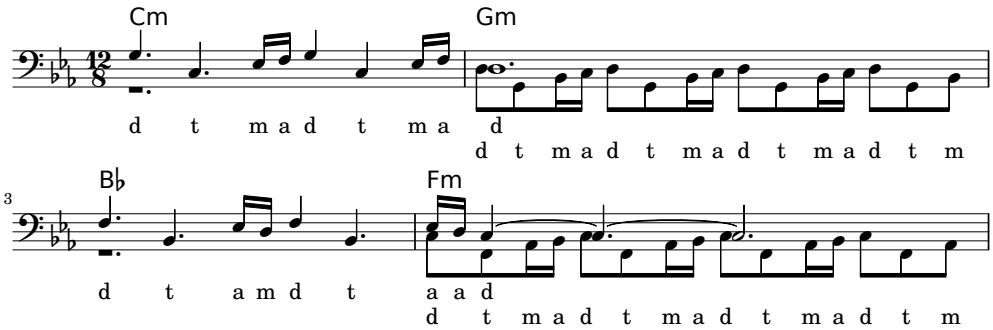

Figure 3: Opening measures from *Game of Thrones*, with manual abstract note annotations. The abstract notes reveal the underlying similarities. The structure of measure 3 is very close to that of measure 1. The structure of measure 4 is identical to that of measure 2.

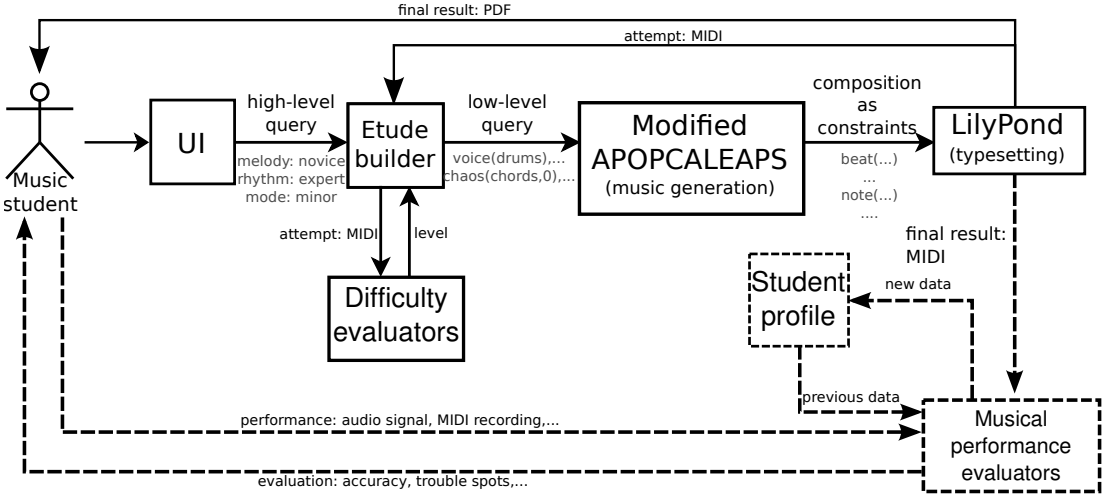

Figure 4: High-level system overview. Dashed lines denote possible future extensions.

4.4 Theme reinstantiation

Thanks to the abstraction described above, there is a conceptually simple way to add a global thematic structure. We can replicate segments with similar features by copying abstract notes from the first instance of a theme to a repeated instance. In this way, the underlying structure of the former is retained, while the concrete notes are generated anew. This transformation generalizes transposition, because newly generated concrete notes can be shifted with respect to previous concrete notes, though this is not always the case. We refer to a repeated theme as a "reinstantiation". Figure 3 illustrates the concept.

Implementation. We represent thematic structure using a new constraint: `theme_boundary(X,Y)`, indicating that theme `X` begins after measure `Y`. Once the first repetition of a theme is computed, it may seem straightforward to copy abstract notes to later repetitions. However, this leads to inconsistencies: repetitions have a problematic status in that they are neither specified nor unspecified. That is, their abstract structure is known but their concrete structure is not. To remedy this, the usual steps of beat generation and abstract note assignment are skipped for these measures. As with normal unspecified measures, however, abstract notes in these measures are instantiated to concrete notes.

4.5 The Big Picture

Figure 4 provides an overview of the system. The dashed lines indicate components which are not in the current prototype, but which we would like to develop in order to obtain a "virtual teacher".

The end user sees a simple interface, indicated by the "UI" box, and sets only a few parameters. Based on the settings supplied to the UI layer, a high-level query is derived for the "Étude builder".

The "Étude builder" can be seen as a high-level generator, which composes music by calling a low-level generator and accumulating parts of its output. It does this by deriving a low-level query which takes into account the current state of the composition (e.g. first measure, last measure), awaiting the result and passing this result to the evaluators. If the result is accepted by all evaluators, the state of the composition is advanced. By evaluating the difficulty of each measure in isolation, it is possible to ignore the absolute length of a composition.

Two shortcomings of our implementation are the cross-language approach and the rather slow computation, on the order of 15 minutes per étude on commodity hardware. A closer integration between the generator and metrics could drastically reduce the time required to generate a composition, but it would require components capable of working with the internal data representations of APOPCALEAPS.

5 Experiments

Perception of difficulty. To determine how closely the metrics outlined above approximate the subjective notion of "difficulty", seven test subjects, including the primary author, sorted a series of generated compositions in order of ascending overall difficulty. For each test subject, a new series of ten compositions was generated. The ten compositions were generated in such a way that they belong to different combinations of a rhythmic and a melodic quartile[1]. To define an "overall" difficulty (according to the metrics) for each composition, we used the product of the rhythmic and the melodic quartile. E.g., for a composition from rhythmic quartile 2 and melodic quartile 3, the overall difficulty is 6. This makes it possible to order the overall difficulty of the ten compositions according to the metrics.

Given a random permutation of such a series of compositions, the expected number of one-element shifts needed to sort the series according to this key is 21.

The subjects' arrangements required an average of 3.86 shifts to achieve the order of the metrics. Also, one test subject had limited experience with sheet music and sorted the pieces very differently from the other subjects. When the results for this subject were disregarded, 1.83 shifts were required.

This demonstrates that the test subjects' arrangements of their compositions were very close to the system's estimate and that the metrics capture a human judgement quite well.

Aesthetic improvements. The subjective notion of aesthetic value was quantified as follows: five subjects were asked to do ten pairwise comparisons between a composition without a theme and a composition with a two-measure theme. The theme occurred in the first two measures; it was repeated in the third and fourth measures; and it reoccurred in the eleventh and twelfth measures. The chord progression was the same in all cases. The melodic and rhythmic quartiles were left unspecified.

In 31 out of 50 comparisons, the composition with themes was preferred. The scope of the experiment was limited, but this is nevertheless a promising result. Furthermore, three subjects pointed out that they preferred some compositions specifically because they noticed variations on a recurring theme.

Inspection of the samples showed that themes are not always easy to recognize: abstract notes may be instantiated in different ways, notes can be joined and the octave for each note is independent from previous variations. Fine-grained control over the instantiation of themes is therefore an interesting subject for future work.

6 Discussion

Related work. Similar applications are described in [8] and [7]. The former work deals with generating études which target specific component skills. In this approach, transformations are applied to existing or stochastically generated pieces of music.

The latter describes two systems designed for sight-reading études. The "Melody Generator" offers great flexibility, but it does not create coherent compositions. The "Harmony Generator" is meant to remedy this by using a global harmonic structure, but it was never fully implemented. Some of the ideas behind the "Harmony Generator" are found in APOPCALEAPS, notably the use of a chord progression.

[1] To obtain somewhat consistent compositions, for each single composition, the rhythmic and melodic quartile could not differ by more than one.

A Virtual Teacher. A goal for future work is to build a full teaching system based on automatic music generation. The dotted lines in Figure 4 indicate components which are still needed to achieve this: performance evaluators would assess aspects such as a student's timing and interpretation. These data could be stored in a profile to track a student's progress and pinpoint areas requiring more work.

7 Conclusions

We have outlined a system demonstrating the possibility of tailoring sheet music to an individual student's skills and goals. This is a promising novel application of music generation. We have also implemented an approach to musical themes which makes études more coherent and realistic.

CHRiSM is particularly well-suited for music generation systems. It is a formalism that inherently maps very well to music composition: it is declarative, it works with symbolic representations, it is based on rules and constraints, it is probabilistic, and it has built-in search and learning mechanisms.

Although the scale of our experiments was limited, an experimental evaluation of our proof-of-concept validates our approach. We hope this work will be a first step towards an exciting new field, which we could call "computational logic for music pedagogy".

References

[1] Gérard Assayag and Shlomo Dubnov. Using factor oracles for machine improvisation. *Soft Computing*, 8(9):604–610, 2004.

[2] John Biles. Genjam: A genetic algorithm for generating jazz solos. In *Proceedings of the International Computer Music Conference*, pages 131–131. International Computer Music Association, 1994.

[3] E Dahlig. Esac database: Essen associative code and folksong database, 1994.

[4] Thom Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1-3):95–138, 1998.

[5] John Ganapes. *Blues You Can Use*. Hal Leonard, 1995.

[6] Jon McCormack. Grammar based music composition. *Complex systems*, 96:321–336, 1996.

[7] Kevin McKee and Boaz Porat. Automated music generation for sight reading, 2008.

[8] Kerstin Neubarth and Tillman Weyde. Using music processing algorithms for exercise generation in music e-learning. In *Axmedis 2006: Proceedings of the 2nd International Conference on Automated Production of Cross Media Content for Multi-channel Distribution...*, page 118. Firenze University Press, 2006.

[9] Lawrence Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[10] Elyse Nicole Reed. The development of sight-reading exercises and procedures for a solfége-based piano curriculum for beginning students. 2012.

[11] Taisuke Sato and Yoshitaka Kameya. Prism: a language for symbolic-statistical modeling. In *IJCAI*, volume 97, pages 1330–1339, 1997.

[12] Jon Sneyers and Danny De Schreye. Apopcaleaps: Automatic music generation with CHRiSM. In *22nd Benelux Conference on Artificial Intelligence (BNAIC'10)*, Luxembourg, 2010.

[13] Jon Sneyers, Wannes Meert, Joost Vennekens, Yoshitaka Kameya, and Taisuke Sato. CHR(PRISM)-based probabilistic logic learning. In M. Hermenegildo, I. Niemelä, and T. Schaub, editors, *26th International Conference on Logic Programming*, Edinburgh, UK, July 2010.

[14] David Temperley. *Music and probability*. The MIT Press, 2007.

[15] Ultimate-guitar.com. Ultimate guitar, 2015.